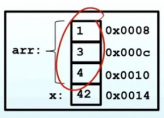
2 Sequential Containers

- Allow for sequential access of elements.
- Maintain the order of elements.
- How can we represent a sequential container?
- One option: store elements contiguously in memory so they are naturally in order.
  - This is how arrays and `std::vector` work.
  - Example:
    ```
    int arr[3] = {1,3,4};
    int x = 42;
    ```



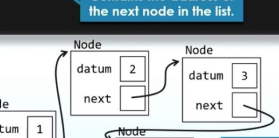
Set vector string

① 有顺序的 container 的形成方式：在 memory 中连续的地址上存储. (array)

3 Using Contiguous Memory

contiguous memory 优点：
1. 不管 array 有多长，找寻第i index的元素所用的时间是相同的
2. 若空间不足，在heap中找寻新的空间复制粘贴，小问题
缺点：
1. 当我们想在array中间插入一个数字的时候，就会很麻烦，因为没有位置。只能将他放在其他的内存位置上。

- Contiguous memory allows indexing through pointer arithmetic, but it has some drawbacks…
- Inserting a new element into the middle of the sequence requires shifting over elements.
- Increasing the capacity requires allocating an entirely new chunk of memory (e.g. `grow()` for `UnsortedSet`).

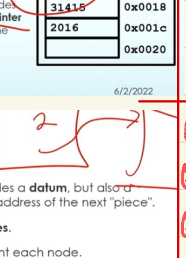


① Contiguous Memory 的优点与缺点.

4 Storing Elements Non-Contiguously

储存数据as pair of value，一般是数据本身，一般是下一个数据的地址。如果是最后一个，放一个Null pointer 告诉compiler这是最后一个数据。

- How can we store a sequence without needing a contiguous chunk of memory?
- We can no longer just move forward one space in memory to get to the next element.
- Instead, we must somehow also keep track of the next element at each point in the list.
- Any ideas for how to do this?
  - Pointers!
  - Each "piece" of the list includes a **datum**, but also a **next pointer** containing the address of the next "piece".



6/2/2022

① datum + pointer 联系前后 node.
② 之前访问 next 方式将不再可用.

5 Nodes

必须要有*，因为，如果没有*，一个Node struct里面就会层层嵌套，包含无数个Node struct。

- Each "piece" of the list includes a **datum**, but also a **next pointer** containing the address of the next "piece".
- We'll call these "pieces" **nodes**.
- Let's use a `struct` to represent each node.
  - Groups together the datum and next pointer.
  - It's "Plain Old Data" (POD). No need for a class.
  - For simplicity, we'll just work with `int`s for now[1].

```
struct Node {
  int datum;
  Node *next;
};
```

Used to store an element of the list.

Contains the address of the next node in the list.

[1] In project 4, you'll write a generic linked list class template.

6/2/2022

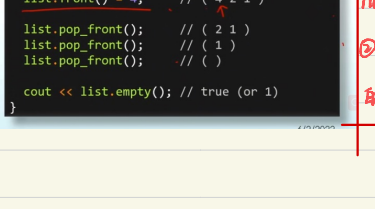
① node 的定义
② node 包含的内容
③ node 是 (POD)
④ node 如何避免层层嵌套.

6 Nodes

```
struct Node {
  int datum;
  Node *next;
};
```

Used to store an element of the list.

Contains the address of the next node in the list.



Use a null pointer (address 0) as a sentinel!

6/2/2022

① node 中含下一个 node 的 address (pointer) (Node * next)
② 如何给末尾 Node 的 next 给值.

7 Linked List Data Representation

```
struct Node {
  int datum;
  Node *next;
};
```

Used to store an element of the list.

Contains the address of the next node in the list.

- Let's also use a `class` to represent an entire list.

```
class IntList {
  Node *first;
};
```


① 创建 list class 来操作整个 list (所有 node)

9 The IntList Interface

```
class IntList {
public:
  // EFFECTS: constructs an empty list
  IntList();

  // EFFECTS: returns true if the list is empty
  bool empty() const;

  // REQUIRES: the list is not empty
  // EFFECTS:  Returns (by reference) the first element
  int & front();

  // EFFECTS: inserts datum at the front of the list
  void push_front(int datum);

  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front();
  ...
};
```

① IntList() 中包含的 Public member function.

10 Using an IntList

这里之所以又可以打印又可以赋值是因为该为front() 的返回值是 int &，所以它返回的是一个memory space，所以就可以进行修改。

```
int main() {
  IntList list;          // ( )
  list.push_front(1);    // ( 1 )
  list.push_front(2);    // ( 2 1 )
  list.push_front(3);    // ( 3 2 1 )

  cout << list.front(); // 3

  list.front() = 4;      // ( 4 2 1 )

  list.pop_front();      // ( 2 1 )
  list.pop_front();      // ( 1 )
  list.pop_front();      // ( )

  cout << list.empty(); // true (or 1)
}
```

Question: Does the outside world need to know about Node?

① 外界不须要知道 Node 的存在.
② int main() 中 list 的使用方式.

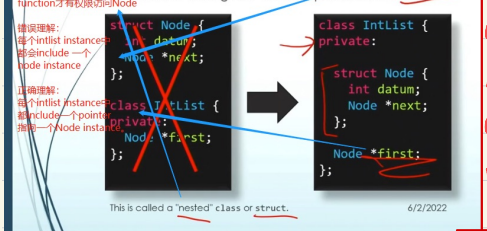
11 Information Hiding

外界不需要知道node的存在，所以可以直接将其放在class的private里。只有class中的member functions. 只有class中的member function才有权限访问Node

错误理解：每个intlist instances都会include一个Node instance

正确理解：每个intlist instances都include一个pointer指向一个Node instance

- Put the Node struct itself inside the IntList class.
- Node can only be used inside the class and its member functions. This is good – it's an implementation detail.



This is called a "nested" class or struct.

6/2/2022

① node structure 定义的位置.
② List class track nodes 的方式
③ IntList() private 中初始化的内容.

12 Implementing IntList: Constructor

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // EFFECTS: constructs an empty list
  IntList() : first(nullptr) { }

  ...
};
```

IntList: first 0

Sets the first pointer to null to indicate an empty list.

① constructor 的实现.

13 Implementing IntList: empty

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // EFFECTS: returns true if the list is empty
  bool empty() const {
    return first == nullptr;
  }

  ...
};
```

IntList: first 0

If the list is empty, the first pointer will be null.

① bool empty(); 的实现方式.

14 Implementing IntList: front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  Returns (by reference) the first element
  int & front() {
    assert(!empty());
    return first->datum;
  }
  ...
};
```

IntList: first

Node: datum 42, next 0

If the list is empty, the first pointer will be null.

① front() 实现
② 先 assert 不然 dereference 一个 nullptr 报错.
③ 返回值是 int &: memory space. (可读值也可修改)

15 Using an IntList

```
int main() {
  IntList list;          // ( )
  list.push_front(1);    // ( 1 )
  list.push_front(2);    // ( 2 1 )
  list.push_front(3);    // ( 3 2 1 )

  cout << list.front(); // 3

  list.front() = 4;      // ( 4 2 1 )

  list.pop_front();      // ( 2 1 )
  list.pop_front();      // ( 1 )
  list.pop_front();      // ( )

  cout << list.empty(); // true (or 1)
}
```

front needs to return an object by reference to support this.

① return an object by reference 方能实现既读又改.

16 Implementing IntList: push_front

我们首先要考虑两种不同的情况：
1. intlist是空的，first指向nullpointer
2. intlist里面有数据，first不指向nullpointer

此2的情况中，我们需要用1个Node都是独立的，这样方便我们后续的操作

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // EFFECTS: inserts datum at the front of the list
  void push_front(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = first;
    first = p;
  }
  ...
};
```

① push_front() 实现
② 分类讨论

16 Implementing IntList: push_front

我们发现，第一种情况和第二种情况下代码是一样的，不需要做特别的区分。

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // EFFECTS: inserts datum at the front of the list
  void push_front(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = first;
    first = p;
  }
  ...
};
```

IntList: first

① 两种情况代码相同.

18 Solution: pop_front

已经通过pointer first delete掉了heap中的最前面的node，但是下一行有调用了已经free的address上的next，是undefined behavior.

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {
    assert(!empty());
    delete first;
    first = first->next;
  }
  ...
};
```

What's wrong with this code?

① pop_front() 最后 顺序关键.
② assert() 是否为空.

19 Solution: pop_front

依然有问题，我们错误的删除了第二个node，并且同时丢失了第一个node的地址

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {
    assert(!empty());
    first = first->next;
    delete first;
  }
  ...
};
```

How about this instead?

① coding 顺序错误的 example.

20 Solution: pop_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // REQUIRES: the list is not empty
  // EFFECTS:  removes the first element
  void pop_front() {
    assert(!empty());
    Node *victim = first;
    first = first->next;
    delete victim;
  }
  ...
};
```

Use a temporary variable to keep track of the Node to be destroyed. Now we can safely change first.

① pop_front() 正确的 implementation.
② temp_value 的使用.

Slides

22 Exercise: Traversing a Linked List

- You can use a pointer to traverse a linked list.
  - Start it pointing to the first `Node`.
  - Move it to each `Node` in turn via `next` pointers.
  - At each step, access the `datum` of the current `Node`.
  - Stop when you get to the null pointer.
- Use this pattern to write a `print` function.

```
class IntList {
public:
  // MODIFIES: os
  // EFFECTS:  prints the list to os
  void print(ostream &os) const {
    // TODO: YOUR CODE HERE
  }
  ...
};
```

① print() 的实现.
② for loop 中使用 node 来进行 traversing
③ List 的 traversing 方式.

23 Solution: Traversing a Linked List

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;

public:
  // MODIFIES: os
  // EFFECTS:  prints the list to os
  void print(ostream &os) const {
    for (Node *np = first; np; np = np->next) {
      os << np->datum << " ";
    }
  }
  ...
};
```

这里不能np++，因为地址不连续。

24 Linked Lists and Dynamic Memory

```
void func() {
  IntList list;
  list.push_front(1);
  IntList list2 = list;
  list2.push_front(2);
}
```

Heap

- Draw a memory diagram. Are there any issues with this code?

① shadow copy 引导致的 Big three 的问题. (heap)

26 IntList Big Three

- Do we need custom implementations of the Big Three for `IntList`?
- Yes. `IntList` owns and manages `Node`s dynamically allocated on the heap.

① IntList 需要 big three.

28 The Big Three

How do we avoid code duplication?

- Destructor
  1. Free resources
- Copy Constructor
  1. Copy regular members from `other`
  2. Deep copy resources from `other`
- Assignment Operator
  1. Check for self-assignment
  2. Free resources
  3. Copy regular members from `rhs`
  4. Deep copy resources from `rhs`
  5. return `*this`

我么可以通过写 helper function 来减少代码重复

① 通过分析 Big three 所执行的功能，找出重复，写 helper function.

28 The Big Three

How do we avoid code duplication?

- Destructor
  1. Free resources — `pop_all()` / `clear()`
- Copy Constructor
  1. Copy regular members from `other`
  2. Deep copy resources from `other` — `push_all()`
- Assignment Operator
  1. Check for self-assignment
  2. Free resources — `pop_all()`
  3. Copy regular members from `rhs`
  4. Deep copy resources from `rhs` — `push_all()`
  5. return `*this`

① pop_all()
② push_all().

29 pop_all and push_all

```
class IntList {
...
private:
  ...

  // EFFECTS: removes all nodes from the list
  void pop_all();

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other);

};
```

① parameter

② .push_all()

parameter

pass by reference

(not copy)

29 pop_all and push_all

```
class IntList {
...
private:
  ...

  // EFFECTS: removes all nodes from the list
  void pop_all();

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other);

};
```

6/2/2022

30 Implementing pop_all

```
class IntList {
...
private:
  ...
  // EFFECTS: removes all nodes from the list
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other);
};
```

一定要在for loop之前先得到size，因为如果在for loop的condition的位置使用size()，就会发现再增加，size()在减少。

6/2/2022

① pop_all的两种实现.

② for loop之前就要得到初始的 size();

38 Implementing pop_back

- What if we want to remove from the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation...

```
struct Node {
  int datum;
  Node *next,
};
```

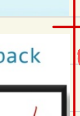

问题是我们无法拿到倒数第二个的地址

6/2/2022

① 欲写 pop_back() 产生的问题

(缺点，private member)

31 Implementing push_all



问题是copy过来之后，会出现倒置

```
class IntList {
...
private:
  ...
  // EFFECTS: removes all nodes from the list
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other) {
    for (Node *np = other.first; np; np = np->next) {
      push_front(np->datum);
    }
  }
};
```

1 2 3

empty 3 2 1

What's wrong with this code?

6/2/2022

① push_all() 也现倒置问题.

39 Implementing pop_back

- What if we want to remove from the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation...

```
struct Node {
  int datum;
  Node *next,
  Node *prev;
};
```


6/2/2022

① 解决方法: IntList class private member variable 中增加 "prev" (node*)

32 Implementing push_all

```
class IntList {
...
private:
  ...
  // EFFECTS: removes all nodes from the list
  void pop_all() {
    while (!empty()) {
      pop_front();
    }
  }

  // EFFECTS: copies all nodes from the other list
  //          to this list
  void push_all(const IntList &other) {
    for (Node *np = other.first; np; np = np->next) {
      push_back(np->datum);
    }
  }
};
```

To avoid a backwards copy, we could use a push_back function.

① 解决方法. push_back()

40 Linked List Template

List.h

```
template <typename T>
class List {
public:
  void push_front(T v);
  T & front();
private:
  struct Node {
    T datum;
    Node *next;
  };
  Node *first;
};
```

```
class List<int> {
public:
  void push_front(int v);
  int & front();
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
};
```

```
class List<Duck> {
public:
  void push_front(Duck v);
  Duck & front();
private:
  struct Node {
    Duck datum;
    Node *next;
  };
  Node *first;
};
```

The compiler instantiates the template as needed according to how it is used in the code.

```
#include "List.h"
int main() {
  List<int> list1;
  List<Duck> list2;
}
```

Member functions of a template must be defined or #included in the .h file, though they need not be defined directly in the class template.

6/2/2022

① List 和 template 相结合.

② List 创建时

注意上方要加:

template <typname T>

③ List 使用时

注意: List<int> list1;

标明该 list 中的 variable type.

34 Implementing push_back

single-linked指的是像只有next，只能向后链接

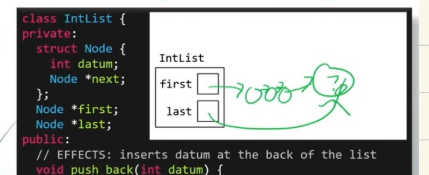
double-linked指的是既有next，又有previous，既能向前又能向后。

single ended指的是只有first，

double ended指的是既有first 又有last

- What if we wanted to insert at the end of the list?
  - We have to traverse all the way from the front!
  - Instead, let's change the data representation...

```
class IntList {
  Node *first;
  Node *last;
};
```



6/2/2022

① 什么是 single-linked single-ended

② 什么是 double-linked double-ended.

③ 需要新增什么 private member variable 来支持 push_back()

35 Implementing IntList: push_back

问题是，如果list之前是空的，则就会出bug

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // EFFECTS: inserts datum at the back of the list
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    last->next = p;
    last = p;
  }
  ...
};
```


① push_back 得分情况讨论.

35 Implementing IntList: push_back

front push_front

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // EFFECTS: inserts datum at the back of the list
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    last->next = p;
    last = p;
  }
  ...
};
```


6/2/2022

② 为什么 push_front() 在只有 "next" 的时候不用分类讨论, 而 push_back() 需要.

36 Implementing IntList: push_back

```
class IntList {
private:
  struct Node {
    int datum;
    Node *next;
  };
  Node *first;
  Node *last;
public:
  // EFFECTS: inserts datum at the back of the list
  void push_back(int datum) {
    Node *p = new Node;
    p->datum = datum;
    p->next = nullptr;
    if (empty()) { first = last = p; }
    else {
      last->next = p;
      last = p;
    }
  }
};
```


① push_back 的实现.